A VERY BASIC GUIDE TO THE USE OF FORTRAN

(This is a preliminary draft and may contain errors -- use with caution)

Richard T. Woodward

I. Basic advice on programming

If you follow these rules your life as a programmer, no matter what language you're using, is likely to be much less aggravating.

1. Have a clear idea of how your program is going to work. If your program involves complicated equations, make sure that you write them down on paper before you touch a keyboard
2. When ever possible work little by little and run your program as you add bits. That way if you there is an error in your program it is very easy to identify where the error is.
3. Avoid using numbers in the body of your program. Parameter values should be specified near the top of your program and then you should call those named parameters in the rest of your program. That way if you want to make a change in a parameter value you only have to do it once.
4. NEVER BLINDLY COPY A PROGRAM. It may look easier when you get started to copy someone else's program and then adopt it to your needs. But you will end up taking a lot less time if you map out your program first and then pull lines from other programs as you identify the need.

II. How a Fortran program works

A Fortran program is an ASCII text file with commands that conform to the specific syntax rules of Fortran. The old version of the rules is called Fortran 77 (this is what I know). The new version of the rules is called Fortran 90 (what I'd like to know).

A Fortran file (e.g. `TEST.FOR`) gets read by a compiler. This is just an ASCII file that can be written in any editor you like. I usually use MSWord, making sure I save as a text file. This text file then gets processed by the compiler. The compiler will normally generate some files after each compilation. The names of the files will normally be generated by the compiler from the source code file name (e.g., `test.obj` and `test.sld`). The one file that you really care about is called the executable (e.g., `TEST.EXE`). The executable file is the program that you actually run.

What we spend most of our time is on how you write the `.FOR` file. How you turn the `.FOR` file into an `.EXE` file depends on what compiler you use. Compilation using the NAG compiler is discussed below.

III. Basic Fortran syntax

A. Spacing on each line

The first space is reserved for commenting out a line, in general use a `*` or a `c` if you don't want that line to be counted

The next 4 spaces can be used for numbering a line, you can then use that number to refer to that line elsewhere in the code.

The sixth space is a continuation maker. If you put a `$` in that space, then the compiler will know that that line is a continuation of the previous line.

Spaces 7-72 are for commands

The compiler will ignore anything beyond the 72nd space.

A comment. Note that even though a commented line could have writing in the spaces 2-6 or after space 72, I leave these spaces blank so that it is easier to see where I am when writing the program.

Here's some examples:

```
*     ----------------------------------------------------------------
*     I use this form to write comments in the code.  The lines above
*     below this comment are exactly 72 characters long. This way I can
*     clearly tell whether or not I've gone beyond 72 characters.  Note
*     also that the * at the beginning is followed by 6 spaces.  That
*     way I can easily see if I'm in the area allowed for writing code.
*     ----------------------------------------------------------------
123456789012345678901234567890123456789012345678901234567890123456789012
```

(note: the numbers below the line above are only intended to demonstrate how spacing should appear in your code and should not appear in an actual program)

A simple command

```
      a = 10.0
12345678901234
```

A numbered line. These are frequently used to move around in the program, for example, if your program has converged, you might want to jump to somewhere else in the program

```
 128  continue
12345678901234
```

A command broken into two lines, note I'm also showing here the exponent function (`**`) and Demonstrating that blank spaces in the spaces 7-72 are of no significance

```
      x27 = a + b *( c - 2.78782376487)**
     $      ((3.8765*a+b)  - 11.98765)
1234567890123456789012345678901234567890123456789012345678901234567890123456789012
```

Although spaces are of no consequence, unnecessary commas or any other symbol will lead to errors and the program will not compile. On the other hand, pay attention to the use of commas, if you forget a comma, your program won't work.

IV. Programs, subroutines and user-defined functions

A program can be made up of many components. It must always contain a main program but that program might *call* subroutines or user-defined functions. In simple programs these are usually contained in one file, although with most compilers it is possible to have them located in separate file. Each of these components is a separate unit introduced with at statement (`Program, Subroutine,` or `Function`) and closed with an `End` statement. Never place a subroutine within your main program -- they must appear after the main program's end statement.

A. The main program

The main program should start with a statement giving the name of the program. The last line indicates the end of the program

```
Program testprog
...
end
```

B. Program structure

Between the beginning of your program and the end there are two very important distinct blocks. The first block is called the initialization block. Here you *initialize* the parameters and variables that you will use in the program (discussed below). The bottom line is that you can't use a variable in your program unless you've introduced the variables in the initialization block.

Following the initialization block is the main part of the program in which you assign values to your variables, make calculations, read input and output etc.

After we've discussed the various components of your programs below, I will suggest a basic structure that you should use to keep your programs orderly and running.

C. Subroutines

As your program becomes more complicated you may want to use subroutines to make your program cleaner and easier to follow. A subroutine must start with a statement indicating the name of the subroutines and the arguments of the subroutine. For example in the subroutine below, x, y and z are input variables and a is the output.

```
subroutine average(a,x,y,z)
real a, x, y, z
a = (x + y + z)/3.0
end
```

Calls to subroutines. In the main program or in other subroutines you can subroutines as follows

```
call average(a,x,y,z)
```

D. Functions

Like subroutines, user-defined functions can be separate little programs that define a function. The key difference is in how it structured. For example we might have a function unit as follows (note the return line).

```
real function averagef(x,y,z)
real x, y, z
averagef = (x + y + z)/3.0
return
end
```

This function would then be called in the main program as follows:

```
a = averagef(x,y,z)
```

You probably shouldn't worry about using subroutines and functions until you have a fair amount of experience.

V. Variables, parameters and arrays

Every bit of data that you work with in Fortran must be a variable or a parameter. Parameters are set at the top of each program unit (main program or subroutine). These cannot be changed. e.g.,

```
parameter(pi = 3.141593)
parameter(n1  = 3, n2 = 7)
```

Variables and parameters are of 3 basic types characters, integers and reals. All variables must be initialized at the beginning of the program unit. You can't actually do anything (e.g., specify a value for a variable or carry out calculations) until all your initializations are complete. See some of the example programs at the end.

Characters variables: When defining a character variable you must define how big it is, e.g.,

```
character*10 direction
```

In this case the `direction` variable might be set equal to "`up`" at one point and "`down`" at other times, These are usually only used to make output prettier or more informative and we won't use character variables very much.

Real variables: A real variable is a numerical approximation of a point on the real number line. The default is an eight decimal approximation. For example, if the variable x is initialized with the statement

```
real x
```

then the statement

```
x = 7.0
```

will really lead to a variable equal to either 7.0000000 or it could be 7.0000001 or even 6.99999999. This is important to remember since `x = 0` might actually lead to a negative number so that $x^{½}$ would be undefined even though $0^{½}$ is perfectly fine. It's a good idea to always use a decimal point when specifying a real variable because some compilers don't correctly process a statement like

```
x = 7
```

If you use double precision variables, then there are 16 digits of precision. These variables are initialized one of two ways:

```
double precision y
real*8 y
```

Double precision and standard precision variables don't get along. For example, if we defined x and y as above, then the statement

```
y = x
```

would almost certainly lead to nonsense. **Be consistent, if you use one double precision variable, make them all double precision**.

Integers: Integers are *not* reals with a bunch of zeros. They are only the digit, e.g.,

```
integer i, j, m
```

A statement made up of all integers will yield an integer result so that

```
x = (i + j)/2
```

would if i = 2 an j = 3, would try to turn x into an integer and would probably lead to nonsense. However,

```
x = (i + j)/2.
```

would yield 1.5 because the right-hand-side now has a real (2.0).

By default, the compiler will recognize any variable or parameter starting with the letters i-n as integers and all others as reals. Hence, if you wanted to use T to define the limit on your discrete time period 0,1,…,T, you'd need to have two lines in your parameter statement

```
integer T
Parameter (T = 50)
```

Multidimensional variables (Arrays):

The variables discussed above were all scalars. We will frequently work with multidimensional arrays. A one-dimensional array will be a vector, e.g.,

```
real x(7)
```

would be a 1 by 7 vector. Each element of the vector could then be called as follows:

```
x(1) = 10.
x(2) = 20.
...
```

One of the frustrating things about Fortran 77 is that it does not manipulate arrays as a whole. For example the lines

```
      parameter (n=7)
      real x(n), y(n)
      x = y
```

will lead to an error statement. We get around this by *looping*. For example

```
      do 102, i = 1, n
      y(i) = x(i)
 102  continue
```

By default an array is indexed starting at 1, but sometimes it will be convenient to start at zero or some other integer. In this case we write

```
      real x(0:7)
```

In this case x has 8 elements, x(0), x(1), …, x(7). Once you've assigned a value to one of the cells of this matrix, e.g.,

```
      i=3
      x(i)=21.6
```

a new variable, say *z*, would be set equal to 21.6 by the commands

```
      i=3
      z=x(i)
```

or, simply

```
      z=x(3)
```

Arrays of more than two dimensions:

Fortran can handle up to seven dimensions. The following command

```
      real x(7,8)
```

would lead to a 7 by 8 matrix,

```
      real x(n1, n2, n3, n4)
```

would be a 4 dimensional object with *ni* point in the i^{th} dimension. Defining such multidimensional arrays can sometimes be convenient. For example, suppose we have a function

Val(x_1, x_2, x_3, x_4) that gives you some number for each value of x_1, x_2, x_3 and x_4, it might be useful to have a four dimensional object V which contains the value of the function at each defined point of the arrays x_1, x_2, x_3 and x_4.

```
      real V(n1, n2, n3, n4), x1(n1), x2(n2), x3(n3), x4(n4)
      ...
      do 111 i1 = 1, n1
      do 111 i2 = 1, n2
      do 111 i3 = 1, n3
      do 111 i4 = 1, n4
      V(i1,i2,i3,i4) = Val(x1(i1),x2(i2),x3(i3),x4(i4))
 111  continue
```

Look carefully at the above loop to see how it hits every possible value of V.

Just like reals, integers and characters can be multidimensional. You can use parameters to specify a dimension and this is a very good habit -- it makes changing your code much easier, e.g.,

```
      parameter(n = 7)
      real x(n)
      integer i(n)
      character textarray(n, 0:n)
```

Implicit variables: If you want to avoid defining all the variables you might use during the course of your program, you can include at the beginning of your code a statement like

```
      implicit   real*8(a-h,o-z)
```

which allows you to use scalar variables without initializing the variable. This can be helpful, but dangerous. My advice is to avoid using implicit statement for the course and then consider using it in your own work should you use Fortran further.

VI. Writing output and reading input

If you want to write a variable or parameter to the screen without any particular format, you would write

```
      write(*,*) x
```

or

```
      print x
```

In the write statement above, the first space inside the parentheses indicates where you want to write the output, *, means to the screen. The second * indicates what format you want to use and * indicates no format.

Format statements allow you to make your output a lot prettier and easier to read. Let's say we want an output file called `stuff.out`, and we want the integer `i` in 3 spaces followed by the variables x, y and z in 8 character blocks with 3 decimals:

```
      real x, y, z
      integer i
      open (unit=11, file='stuff.out')
      ...
      write (11,1234) i, x, y, z
 1234 format(i3, 3f8.3)
```

There are, therefore, three important lines that play a role in writing output.
First is the definition of where your output is going to be put and the unit number associated with that file. The unit number can be any number between 1 and 99, but it's a pretty good idea to keep your numbers greater than 10 since some lower numbers are reserved on some systems. As written above the file would be written (or written over) in the directory from which you run the program. If you want to write it somewhere else, simply define that, e.g.,

```
file='c:\642\stuff.out'
```

The second important line is the line where you tell the computer to write the output. All variables should be printed out, each separated by a comma. The elements of multidimensional arrays are printed out in the in sequence so that if you have an array `x(2,2)` and you introduce

the command `write(*,*) x`, then elements of x will print in the following order `x(1,1),x(2,1),x(1,2),x(2,2)`.

The third piece of writing output is a format statement in which you specify what your output will look like. Formatting is very tricky, can easily lead to syntax errors, and can take a lot of your time. Hence, I recommend keeping your formatting as simple as possible (usually just eliminating the format statement and using a *). Nonetheless, here are some of the key elements of formatting statements.

The format statement begins with a line number, to which you refer when you want to invoke that format. Inside the parentheses you define any text you want to print out, in single quotation marks, followed by the format for each variable that will be printed. For example, with i3 = 12, x=4.0, y=5000.0 and z = 6.127777, the statements

```
      write (*,1235) i3, x, y, z
 1235 format('i=', i3, 'x, y & z =', 3f8.3)
```

would yield

```
 i= 12 x, y & z =    4.000  5000.000     6.127
```

It's important to be careful about what gets printed where. If you try to print a real variable as an integer or vice versa you will get an error. Consult a more detailed programming guide if you want to learn more about formatting.

The format statements can appear anywhere in your program after the initialization block, but I usually put them at the bottom of the code, just before the `end` statement to keep things organized.

In theory, reading from a text file or the keyboard (*) is just like writing, only backwards. But this can be tricky so be very careful when you do it. Again, consult a more detailed programming guide if you want to learn more about this.

VII. Basic Functions

There are a bunch of implicit functions that are standard in Fortran 77. In addition to =, –, * and /, most of the ones we will use are in the following list:

`abs(x)` = $|x|$
`exp(x)` = e^x
`log(x)` = $ln(x)$
`real(i)` - converts an integer, i, to a real number
`max(x,y)` - self explanatory
`min(x,y)` - self explanatory
`x**n` = x^n
`sqrt(x)` = $x^{½}$

Other functions might be contained in libraries to which your compiler automatically links. For example, there might be an average function that is essentially like the one we have above in the

library. Ones that you might want to use are include random number generators or date&time functions. There are no libraries associated with the Nag compiler that is available in the grad lab.

If/Then statements

sometimes you can use an if statement in one line

```
if(x .gt. 0.0) x = x**.5
```

however, if you have a number of things to do, then you have to use an if( ) then statement

```
if(x .gt. 0.0 .and. y .gt. 0.0) then
   x = (x**.5)*(y**.5)
   i = 1
endif
```

You can also get quite complicated, …

```
if(x .gt. 0.0 .and. i .eq. 2) then
   x = (x**i)
   i = 1
else
   x = 0.0
   i = 0
endif
```

Similarly, you can have

```
if( ... ) then
    ...
elseif( ... ) then
    ...
elseif( ... ) then
    ...
endif
```

You can use the following relational statements in If statements: `.lt. .gt. .eq. .and. .or.`

Goto statements

You will frequently want to be able to jump to some other place in your program. For example, once part of your program has converged, it will be useful to jump out of the loop to a place where you go on to the next part of your program. You do this using numbered lines.

If the criteria for jumping out can be captured in one statement, then this be accomplished by a statement like

```
if( ... ) goto 999
```

Other times you'll need to use an if/then statement:

```
if( ... ) then
    ...
    goto 999
```

VIII. Looping

In numerical analysis we are constantly looping -- looping across elements of arrays, looping to repeat operations, looping to gradually find the numerical solution. Get used to it, it's very important.

A. Looping to sum:

Suppose you want to set a variable *y* equal to the sum across the elements of an array *x* that is of dimension *n*. This would be accomplished using the following

```
      parameter (n=20)
      real x(n), y
          ...
      y=0
      do 853, i=1, n
      y = x(i)+y
 853  continue
```

note that it is very important to start *y* at zero because otherwise each time you implemented the loop you would add to the previous sum.

B. Looping to multiply:

If you wanted to take the product of all the elements of *x*, the process would be virtually the same but you would initialize *y* at 1 instead of 0.

A tip: When you're initially writing your program, write to the screen some of your output inside each loop to make sure that you're actually doing what you want to do.

IX. Debugging

Debugging a program is a tedious and difficult process. The process takes place in two stages that repeat over and over again as you write the program.

1. The first stage is what I call compilation debugging - getting the syntax correct so that your compiler understands your code.
2. The second stage is what I call logic debugging - getting your program to not only run, but do what you want it to.

The key to debugging your code is to build your program piece by piece, making sure that each piece runs as you intend it to before moving on. Writing a large program and then discovering that you have 50 errors is a terrible waste of time. Write a little bit, run it, fix any errors and only then move on. Even very experienced programmers make syntax errors or errors in logic, you will make lots of them as you learn.

Compilation debugging.
If you've written a program and it doesn't compile you need to find your syntax error(s). Most compilers will at least tell you the line on which it had difficulty and a typically cryptic message as to the nature of your error. Look at these messages carefully, they probably will tell you what your problem is. If you can't figure out where your problem is you might want to start by simply commenting out the line indicated. If the program then compiles, you know where your error is.

Logic debugging
After your program compiles your real work is to begin, making sure that the answers you're getting are what you want to get. The key to this is to think through your problem carefully,

write it out with a pencil before you ever touch the keyboard. Don't be afraid to write out lots of output as you work through your program. In my programs I frequently write:

```
*     ----------------------------------------------------------------
*     The next three lines are for debugging
*     ----------------------------------------------------------------
      do 771, i=1,n
      write (11,*) x(i), y(i)
 771  continue
```

Sometimes your program will crash and you can't determine where. In this case one trick is to place a series of markers throughout your program that write some output when the program gets to that point. A marker would take the form:

```
      write (11,*) 'mark 1, i = ', i
```

X. An outline of a simple Fortran 77 program

I. Open with a program statement

```
*     ----------------------------------------------------------------
      Program testprog
*     ----------------------------------------------------------------
```

II. Then initialize your parameters and variables.
If you use an `implicit` statement it should come here too.

```
*     ----------------------------------------------------------------
      parameter(pi = 3.141593)
      parameter(n  = 3)
      real a, x(n), y
      character*10 currentdir
      integer i, j, m
*     ----------------------------------------------------------------
```

III. Then open the files to which you'll be writing output

```
*     ----------------------------------------------------------------
      open (unit=11, file='stuff.out')
*     ----------------------------------------------------------------
```

IV. Then comes the body of your program

```
*     ----------------------------------------------------------------
      a = 4.
      y = pi*a
      do 123, i=1, n1
      x(i) = y
      y = y* y
 123  continue
*     ----------------------------------------------------------------
*     Now we write some stuff to the screen
*     ----------------------------------------------------------------
      write (*, 1001) y, a, n1+1
      write (*, 1002) a, x
      write (*, 1003)
*     ----------------------------------------------------------------
*     Then we write it again to your output file, stuff.out
*     ----------------------------------------------------------------
      write (11, 1001) y, a, n1+1
      write (11, 1002) a, x

*     ----------------------------------------------------------------
```

V. I usually put my format statements at the bottom of the program

```
*     ----------------------------------------------------------------
 1001 format (f14.2, ' = ', f4.1, '* pi to the ', i3,'th power.')
 1002 format (f4.1, '* pi to 1st, 2nd and 3rd power = ',f4.1,f6.1,f8.1)
 1003 format ('Now look at the file stuff.out')
*     ----------------------------------------------------------------
```

VII. Then end your program

```
*     ----------------------------------------------------------------
      end
```

XI. Using the NAG Fortran compiler

The NAG 32-bit Fortran compiler will be available in the grad-student computer lab. It is also available elsewhere on the Texas A&M campus, probably also including the ACC on the first floor of the Blocker Building. For the present, the only instructions on the use of NAG compilers that I have are from a similar 16-bit compiler. These are available at http://agecon.tamu.edu/ageco/faculty/woodward/642/NagNotes.htm.

XII. Example

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*     ---------------------------------------------------------------------
*     Program Celsius Table: Prints simple Fahrenheit-Celsius table
*     ---------------------------------------------------------------------
      program celsius_table

      implicit none
      real Fahrenheit, Celsius
      integer i
*     ---------------------------------------------------------------------
*     End of initialization block
*     ---------------------------------------------------------------------

*     ---------------------------------------------------------------------
*     Write a couple of title lines to the screen
*     ---------------------------------------------------------------------
      write(*,*) '  Fahrenheit     Celsius'
      write(*,*) '-------------------------'

*     ---------------------------------------------------------------------
*     Start of loop defining Fahrenheit from 0 to 250 and then
*     calculating the equivalent in Celsius
*     ---------------------------------------------------------------------
      Fahrenheit 0.0
      do 123, i = 1, 25
            Fahrenheit Fahrenheit+10
            Celsius = (5.0/9.0) * (Fahrenheit-32.0)
            write(*,1111) Fahrenheit, Celsius
 123  continue

*     ---------------------------------------------------------------------
*     Formatting line
*     ---------------------------------------------------------------------
 1111 format(F13.0,F12.3)

*     ---------------------------------------------------------------------
*     End of program
*     ---------------------------------------------------------------------
      end
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

XIII. Practice problems

1. What number would the following lines of code yield?

```
      x = 4.0
      do 83, i = 1, 3
 83   x = x*x
```

2. Write a program that writes the variable i to the screen as it counts from $i = 1$ to $i = n$ with n=10.

3. Write a program that defines two vectors, x_1 and x_2 of dimension n_1 and n_2 with n_1=n_2=10. Set the elements of x_1 and x_2 equal to 0.1, 0.2, 0.3, …. Write the vectors x_1 and x_2 to the screen.

4. Write a program that defines two vectors, x_1 and x_2 of dimension n_1 and n_2 with n_1=n_2=10. Set the elements of x_1 equal to $1/xmax_1$, $2/xmax_1$, $3/xmax_1$,… and x_2 = $1/xmax_2$, $2/xmax_2$, $3/xmax_2$,… with $xmax_1$= 3 and $xmax_2$=2.

5. Write a program to find the utility ($u = x_1^{\alpha}x_2^{\beta}$) at each point in for two arrays, x_1 and x_2 that make up an evenly spaced grid between (x1min, x2min) and (x1max, x2max), with n1 and n2 points in each dimension respectively. Write the results at each point both to the screen and to an output file.

6. Write a program to find the maximum utility ($u = x_1^{\alpha}x_2^{\beta}$) subject to the budget constraint $p_1x_1+p_2x_2 \le m$. x_1 and x_2 should be n1- and n2-dimensional arrays. Write the solution (x_1, x_2, u) and the parameters, m, p_1, p_2, α and β to an output file.
   - how does the numerical solution compare to the correct analytical solution? How does the precision of the numerical results change as you change n1 and n2?
   - loop over a variety of parameter values to show how the results change.
   - for small changes in the parameter values does the solution always change?

7. Partial derivatives can be approximated numerically by using the difference approximation $\partial u/\partial x \approx [u(x_1,x_2)-u(x_1',x_2)]/(x_1-x_1')$. Calculate the Lagrange multiplier at each point in the state space.

8. At the optimum for one set of parameter values, numerically verify the saddle point inequality of constrained optimization, i.e.
   $L(x,\lambda^*) \le L(x^*,\lambda^*) \le L(x^*,\lambda)$ for all x and for all $\lambda \ge 0$.
   where $L(x,\lambda)$ is the Lagrangian of the constrained optimization problem,
   maxx u(x) s.t. $g(x) \le 0$. i.e.,
   $L(x,\lambda) = u(x) + \lambda g(x)$
   and x^* and λ^* are the optimal values of x and l respectively.